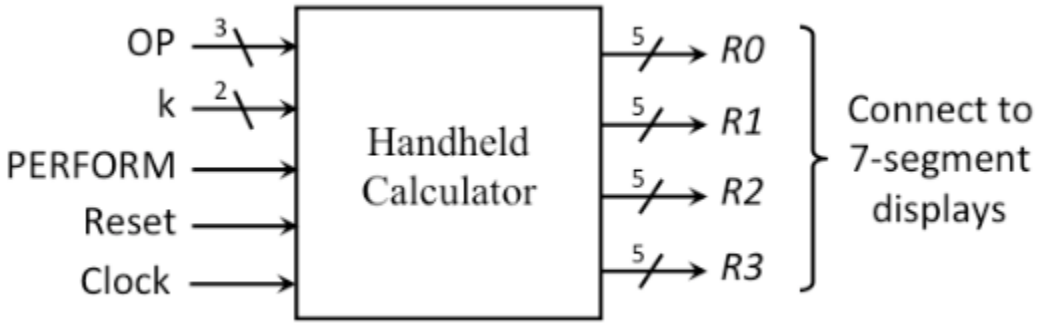
Kyle Thompson

CprE 281: Digital Logic

Final Project: Handheld Calculator

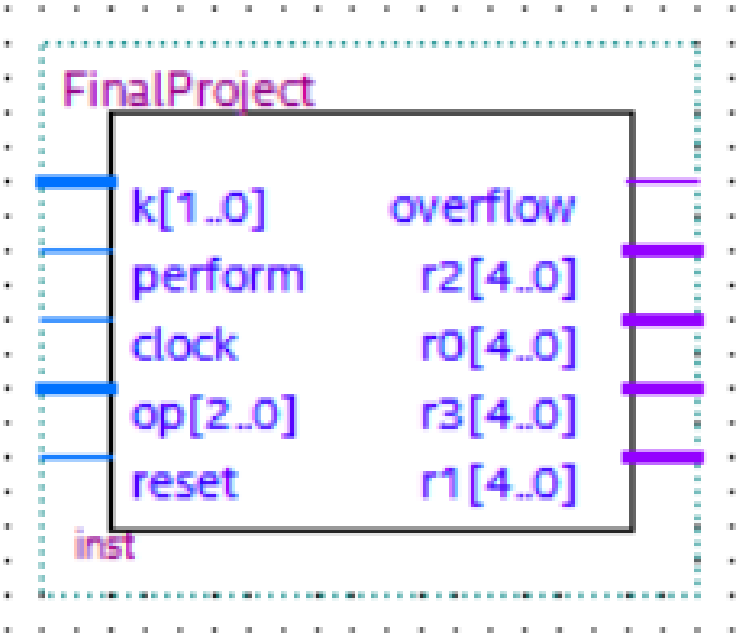
My project was the handheld calculator from a previous semester. My project has 4 registers (R0-R3), each of which hold a 5-bit unsigned integer. There is a 3-bit input named OP (operation), which specifies what operation needs to be performed and a 2-bit input named k which specifies a parameter with a value between 0 and 3 (00 and 11). The signal perform will be controlling when the specified operation is done. The reset signal will be used to set all registers back to 0 and set the FSM back to its initial state. I will be using switches to control my input parameters and I'll be using seven segment displays to show what is stored in my registers. I have also elected to include overflow via a red LED which is calculated before perform is flipped.



(Visual description of input and output)

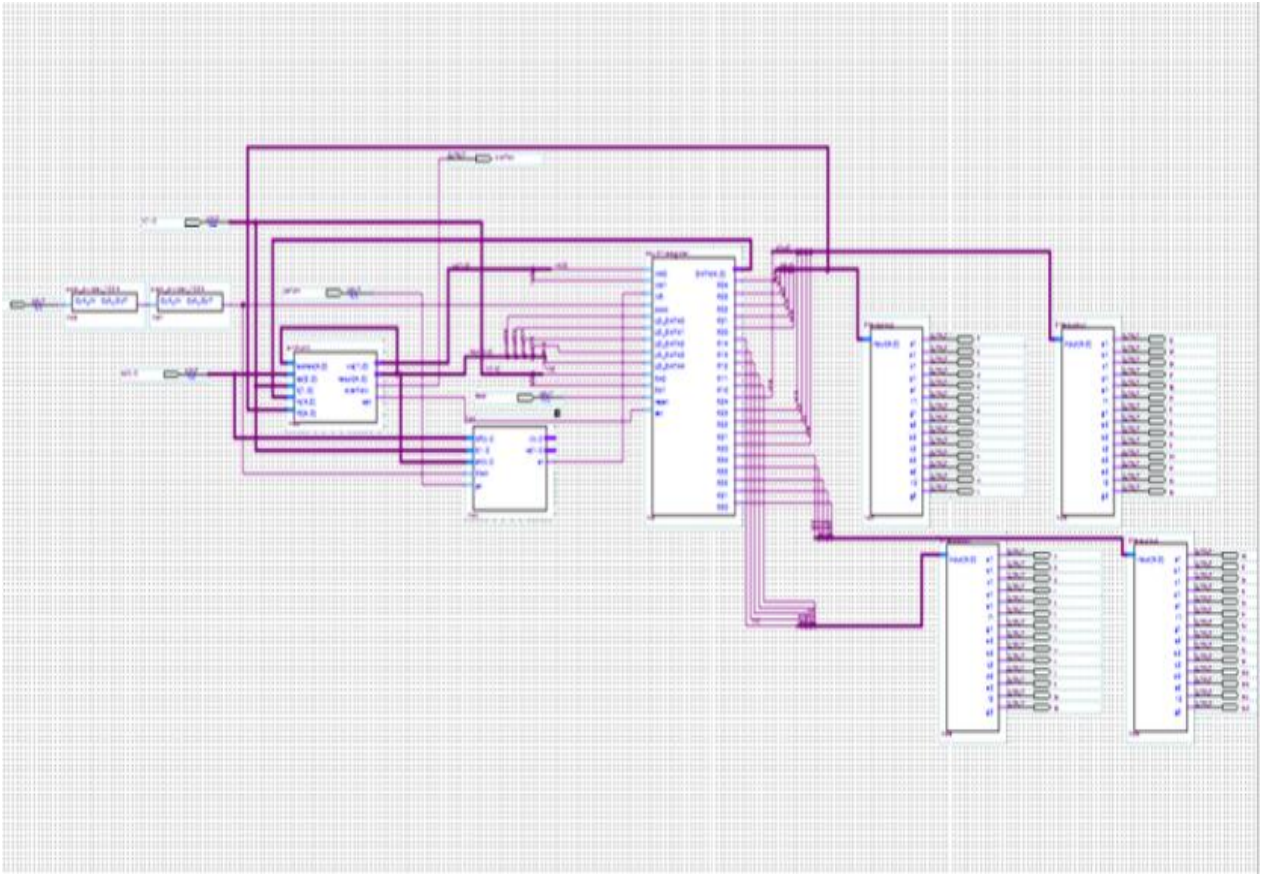
OP	Operation
000	$R0 \leftarrow 0, R1 \leftarrow 1, R2 \leftarrow 2, R3 \leftarrow 3$
001	$R0 \leftarrow k$
010	$R0 \leftarrow Rk$
011	$Rk \leftarrow R0$
100	$R0 \leftarrow R0 + Rk$
101	$R0 \leftarrow R0 - Rk$
110	$R0 \leftarrow R0 \times Rk$
111	$R0 \leftarrow 2^{Rk}$

(Description of basic operations that the calculator must be able to perform)



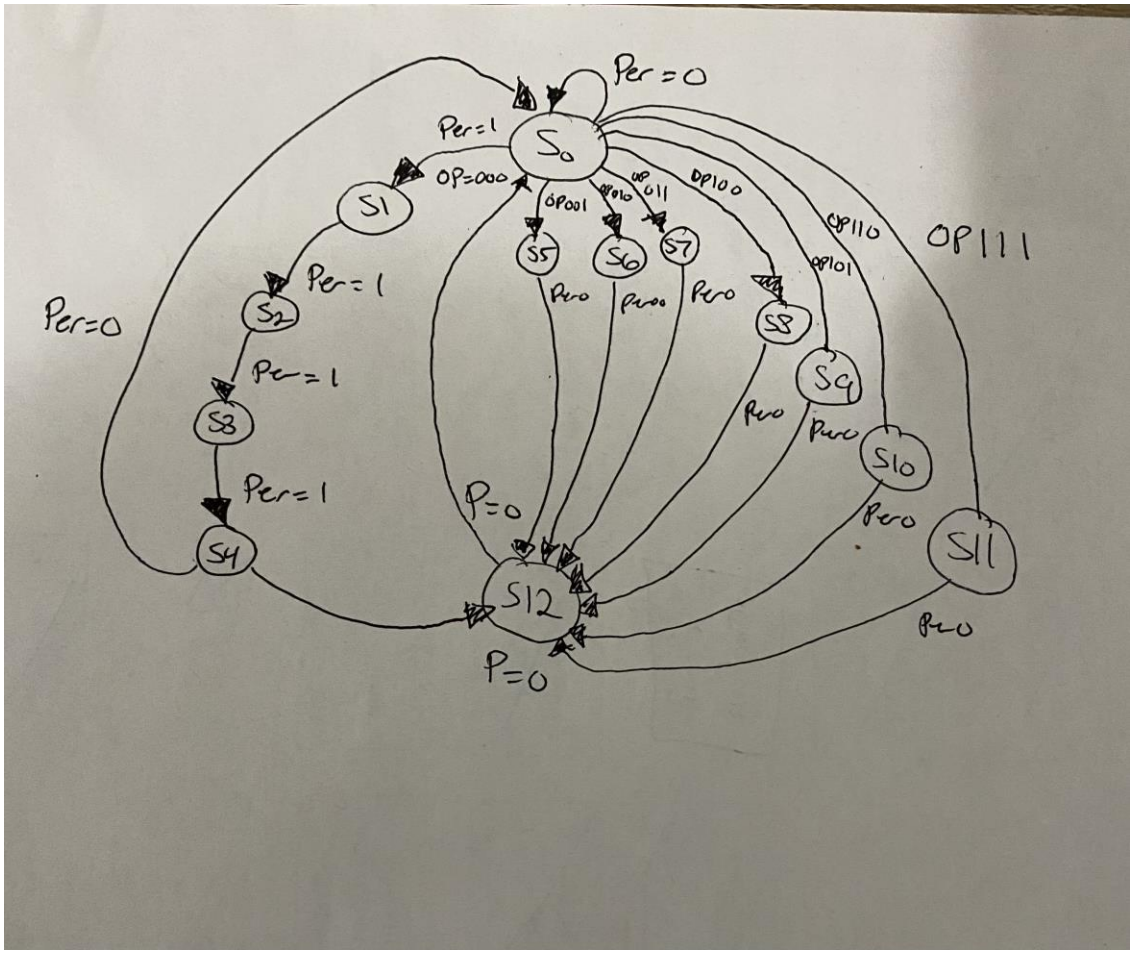
(Showing the result of my project as a final .bdf, my project also included overflow)

For my final project, I included a clock divider that was provided earlier in the semester for our labs. I also created a math unit, which was a part that did all the calculations required for the project as listed in the figure above. I made 4, 5-bit registers files and 2 hexout displays. The FSM I created was used to control the enable on the register file. In this report I will be listing how each of these parts were created and what their purpose was to reach this final product.



State Machine:

I used Verilog code to describe the 13 different parameters produced by this state machine. All the if/else statements control the selected module. In the second part, when the clock edge is positive, based on the module chosen the output will be decided. For this assignment I used the enable output because my math unit was able to control other inputs.



```
module fsm(OP,k,arit,Clock,r,wa,per,en);
input[4:0] arit;
input[2:0] OP;
input[1:0] k;
input Clock, per;
output[4:0] r;
output[1:0] wa;
output en;
reg[3:0] y,Y;
reg[4:0] r;
reg[1:0] wa;
reg en;
parameter[3:0] A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100, F = 4'b0101,
G = 4'b0110, H = 4'b0111, I = 4'b1000, J = 4'b1001, K = 4'b1010, L = 4'b1011, M = 4'b1100;

always@(per,y,OP)begin
   case(y)
      A:if(per)
         begin
            if(OP == 3'b000)Y = B;
            if(OP == 3'b001)Y = F;
            if(OP == 3'b010)Y = G;
            if(OP == 3'b011)Y = H;
            if(OP == 3'b100)Y = I;
            if(OP == 3'b101)Y = J;
            if(OP == 3'b110)Y = K;
            if(OP == 3'b111)Y = L;
         end
         else Y = A;
      B:if(per)Y = C;
         else Y = A;
      C:if(per)Y = D;
         else Y = A;
      D:if(per)Y = E;
         else Y = A;
      E:if(~per)Y = A;
         else Y = M;
      F:if(~per)Y = A;
         else Y = M;
      G:if(~per)Y = A;
         else Y = M;
      H:if(~per)Y = A;
         else Y = M;
      I:if(~per)Y = A;
         else Y = M;
      J:if(~per)Y = A;
         else Y = M;
      K:if(~per)Y = A;
         else Y = M;
      L:if(~per)Y = A;
         else Y = M;
      M:if(~per)Y = A;
         else Y = M;
      default Y = 4'bxxxx;
   endcase
end
   always@(posedge Clock)begin
      y<= Y;
      case(y)
         B:
         begin
```

```
62                  r[4] = 0;
63                  r[3] = 0;
64                  r[2] = 0;
65                  r[1] = 0;
66                  r[0] = 0;
67                  wa[1] = 0;
68                  wa[0] = 0;
69                  en = 1;
70              end
71              C:
72              begin
73                  r[4] = 0;
74                  r[3] = 0;
75                  r[2] = 0;
76                  r[1] = 0;
77                  r[0] = 1;
78                  wa[1] = 0;
79                  wa[0] = 1;
80                  en = 1;
81              end
82              D:
83              begin
84                  r[4] = 0;
85                  r[3] = 0;
86                  r[2] = 0;
87                  r[1] = 1;
88                  r[0] = 0;
89                  wa[1] = 1;
90                  wa[0] = 0;
91                  en = 1;
92              end
93              E:
94              begin
95                  r[4] = 0;
96                  r[3] = 0;
97                  r[2] = 0;
98                  r[1] = 1;
99                  r[0] = 1;
100                 wa[1] = 1;
101                 wa[0] = 1;
102                 en = 1;
103             end
104             F:
105             begin
106                 r[4] = 0;
107                 r[3] = 0;
108                 r[2] = 0;
109                 r[1] = k[1];
110                 r[0] = k[0];
111                 wa[1] = 0;
112                 wa[0] = 0;
113                 en = 1;
114             end
115             G:
116             begin
117                 r[4] = arit[4];
118                 r[3] = arit[3];
119                 r[2] = arit[2];
120                 r[1] = arit[1];
121                 r[0] = arit[0];
122                 wa[1] = 0;
```

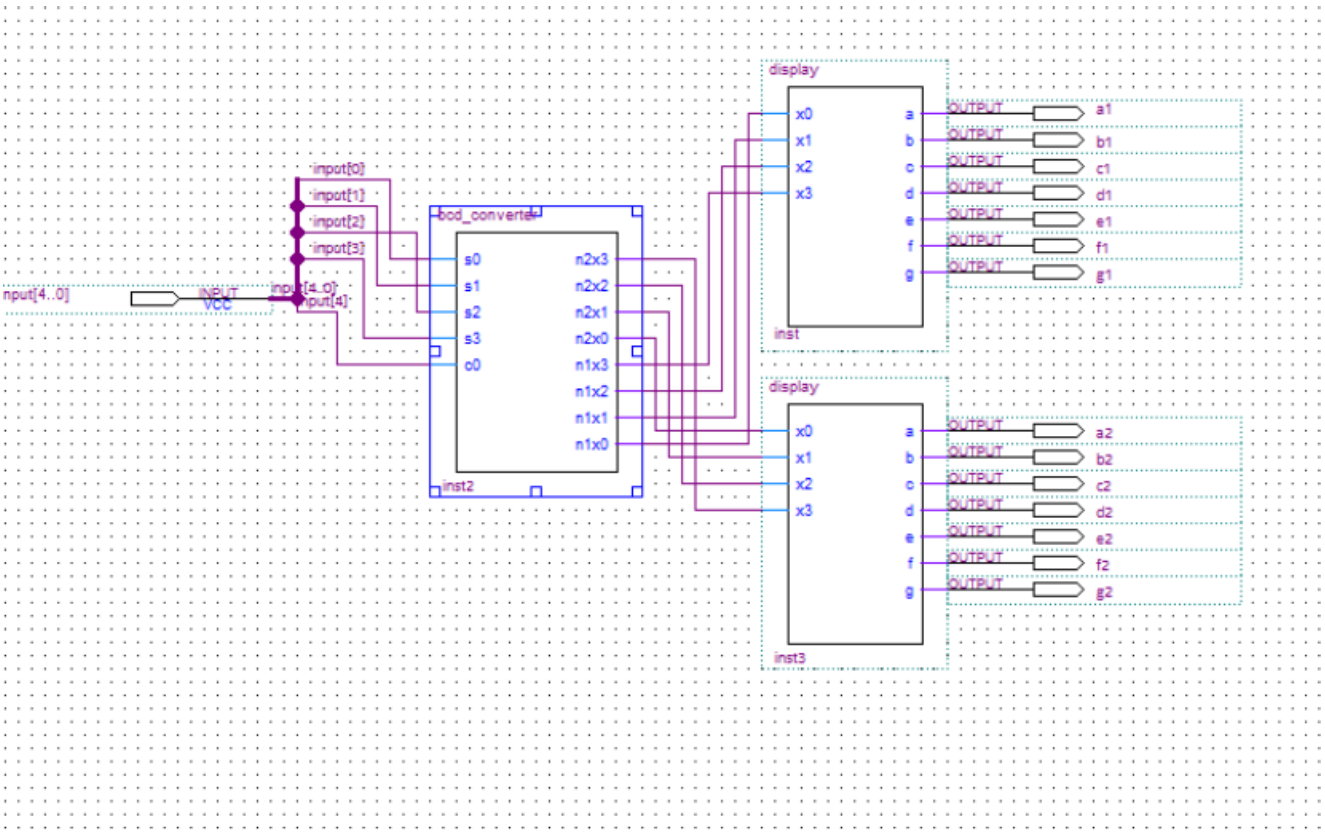
```
123             wa[0] = 0;
124             en = 1;
125         end
126         H:
127         begin
128             r[4] = arit[4];
129             r[3] = arit[3];
130             r[2] = arit[2];
131             r[1] = arit[1];
132             r[0] = arit[0];
133             wa[1] = k[1];
134             wa[0] = k[0];
135             en = 1;
136         end
137         I:
138         begin
139             r[4] = arit[4];
140             r[3] = arit[3];
141             r[2] = arit[2];
142             r[1] = arit[1];
143             r[0] = arit[0];
144             wa[1] = 0;
145             wa[0] = 0;
146             en = 1;
147         end
148         J:
149         begin
150             r[4] = 0;
151             r[3] = 0;
152             r[2] = 0;
153             r[1] = 0;
154             r[0] = 0;
155             wa[1] = 0;
156             wa[0] = 0;
157             en = 1;
158         end
159         K:
160         begin
161             r[4] = arit[4];
162             r[3] = arit[3];
163             r[2] = arit[2];
164             r[1] = arit[1];
165             r[0] = arit[0];
166             wa[1] = 0;
167             wa[0] = 0;
168             en = 1;
169         end
170         L:
171         begin
172             r[4] = arit[4];
173             r[3] = arit[3];
174             r[2] = arit[2];
175             r[1] = arit[1];
176             r[0] = arit[0];
177             wa[1] = 0;
178             wa[0] = 0;
179             en = 1;
180         end
181         M:
182         begin
183             r[4] = 0;
```

```
184                 r[3] = 0;
185                 r[2] = 0;
186                 r[1] = 0;
187                 r[0] = 0;
188                 wa[1] = 0;
189                 wa[0] = 0;
190                 en = 0;
191             end
192         endcase
193     end
194 endmodule
195
```

2Hexout:

In my project I use 2Hexout to display the contents of my registers. It starts by taking in the decimal values of the registers and outputs display values. These are shown via two hex displays on the classroom circuit boards. I will also include the code for my BCD converter and display.

The main purpose of the BCD was to convert a binary input into two outputs for the display to show. The purpose of the display was to take in four binary values and output to the hex display.



(.bdf for 2Hexout)

```
module bcd_converter(s0,s1,s2,s3,c0,n2x3,n2x2,n2x1,n2x0,n1x3,n1x2,n1x1,n1x0);
   input s0,s1,s2,s3,c0;
   output n2x3,n2x2,n2x1,n2x0,n1x3,n1x2,n1x1,n1x0;
   assign n2x3 = 0;
   assign n2x2 = 0;
   assign n2x1 = c0&s2|c0&s3|c0&s3&s2&s1&~s0|c0&~s3&s2&s1&s0;
   assign n2x0 = ~c0&s3&s1|~c0&s3&s2|s3&s2&s1|c0&~s3&~s2|c0&s3&s2&s1&~s0;
   assign n1x3 = ~c0&s3&~s2&~s1|c0&~s3&~s2&s1|c0&s3&s2&~s1;
   assign n1x2 = ~c0&~s3&s2|~c0&s2&s1|c0&~s2&~s1|c0&s3&~s2;
   assign n1x1 = ~c0&~s3&s1|~s3&s2&s1|~c0&s3&s2&~s1|c0&~s3&~s2&~s1|c0&s3&~s2&s1|c0&~s3&s2&s1&s0;
   assign n1x0 = s0;
endmodule
```

(Code for BCD converter)

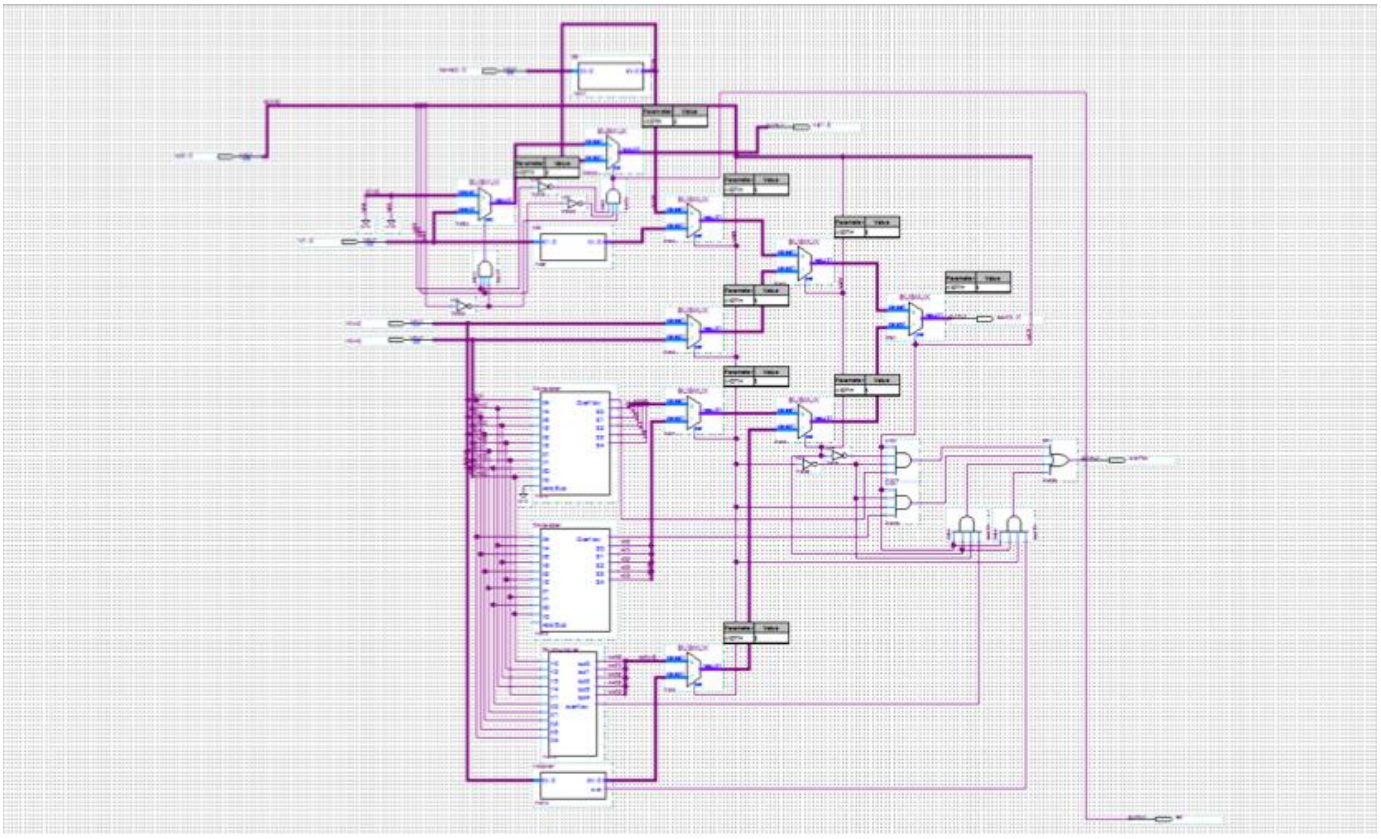
```
module display(a,b,c,d,e,f,g,x0,x1,x2,x3);
   input x0,x1,x2,x3;
   output a,b,c,d,e,f,g;
   assign a = (~x3&~x2&~&x1&x0)|(~x3&x2&~x1&~x0)|(x3&x2&~x1&x0)|(x3&~x2&x1&x0);
   assign b = (~x3&x2&~x1&x0)|(~x3&x2&x1&~x0)|(x3&~x2&x1&x0)|(x3&x2&~x1&~x0)|(x3&x2&x1&~x0)|(x3&x2&x1&x0);
   assign c = (~x3&~x2&x1&~x0)|(x3&x2&~x1&~x0)|(x3&x2&x1&~x0)|(x3&x2&x1&x0);
   assign d = (~x3&~x2&~x1&x0)|(~x3&x2&x1&x0)|(x3&~x2&x1&~x0)|(x3&x2&x1&x0)|(~x3&x2*~x1&~x0);
   assign e = (~x3&~x2&~x1&x0)|(~x3&~x2&x1&x0)|(~x3&x2&~x1&~x0)|(~x3&x2&~x1&x0)|(~x3&x2&x1&x0)|(x3&~x2&~x1&x0);
   assign f = (~x3&~x2&~x1&x0)|(~x3&~x2&x1&~x0)|(~x3&~x2&x1&x0)|(~x3&x2&x1&x0)|(x3&x2&~x1&x0);
   assign g = (~x3&~x2&~x1&~x0)|(~x3&x2&x1&x0)|(x3&x2&~x1&~x0)|(~x3&~x2&~x1&x0);
endmodule
```

(Code for display)

Math Unit:

My math unit is the section where the output is calculated based on operation. The inputs are the current operation (OP), current K, the K register needed, the last result and the current R0 (register 0 value). It outputs the new value, the WA, set and overflow!

In this system I have files doing subtraction, addition, multiplication, a file named ZZZ for OP000 and ZZ0 for OP001. There is also another file named two-power for OP111 (see chart in introduction for example).



(Math unit .bdf)

ZZ0:

Controls the value of OP001 that is output, takes in K and outputs a register value based on it.

```
1   module zzo(k,r);
2      input[1:0] k;
3      output[4:0] r;
4      assign r[0] = k[0];
5      assign r[1] = k[1];
6      assign r[2] = 0;
7      assign r[3] = 0;
8      assign r[4] = 0;
9   endmodule
```

ZZZ:

Controls the value of OP000, the input is the last result, and the next result becomes the output.

```
1   module zzz(r,o);
2      input[4:0] r;
3      output[4:0] o;
4      assign o[0] = ~r[4]&~r[3]&~r[2]&~r[1]&~r[0]|~r[4]&~r[3]&~r[2]&r[1]&~r[0];
5      assign o[1] = ~r[4]&~r[3]&~r[2]&~r[1]&r[0]|~r[4]&~r[3]&~r[2]&r[1]&~r[0];
6      assign o[2] = 0;
7      assign o[3] = 0;
8      assign o[4] = 0;
9   endmodule
```

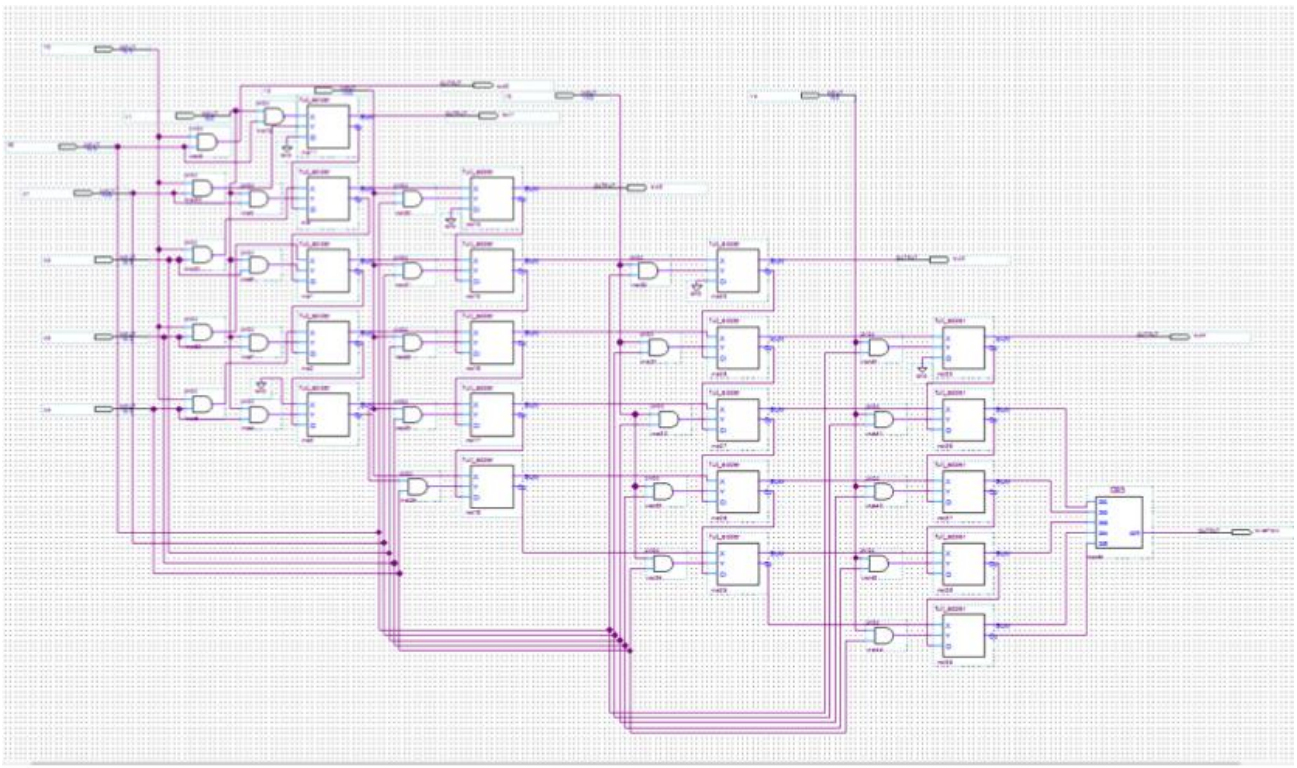
Two-power:

Produces an output for OP111 by inputting register values then returning the new register with overflow.

```
module twopower(r,o, over);
   input[4:0] r;
   output[4:0] o;
   output over;
   assign o[0] = ~r[4]&~r[3]&~r[2]&~r[1]&~r[0];
   assign o[1] = ~r[4]&~r[3]&~r[2]&~r[1]&r[0];
   assign o[2] = ~r[4]&~r[3]&~r[2]&r[1]&~r[0];
   assign o[3] = ~r[4]&~r[3]&~r[2]&r[1]&r[0];
   assign o[4] = ~r[4]&~r[3]&r[2]&~r[1]&~r[0];
   assign over = r[4]|r[3]|r[2]&r[1]&~r[0]|r[2]&r[1]&r[0]|~r[4]&~r[3]&r[2]&~r[1]&r[0];
endmodule
```

5-bit multiplier:

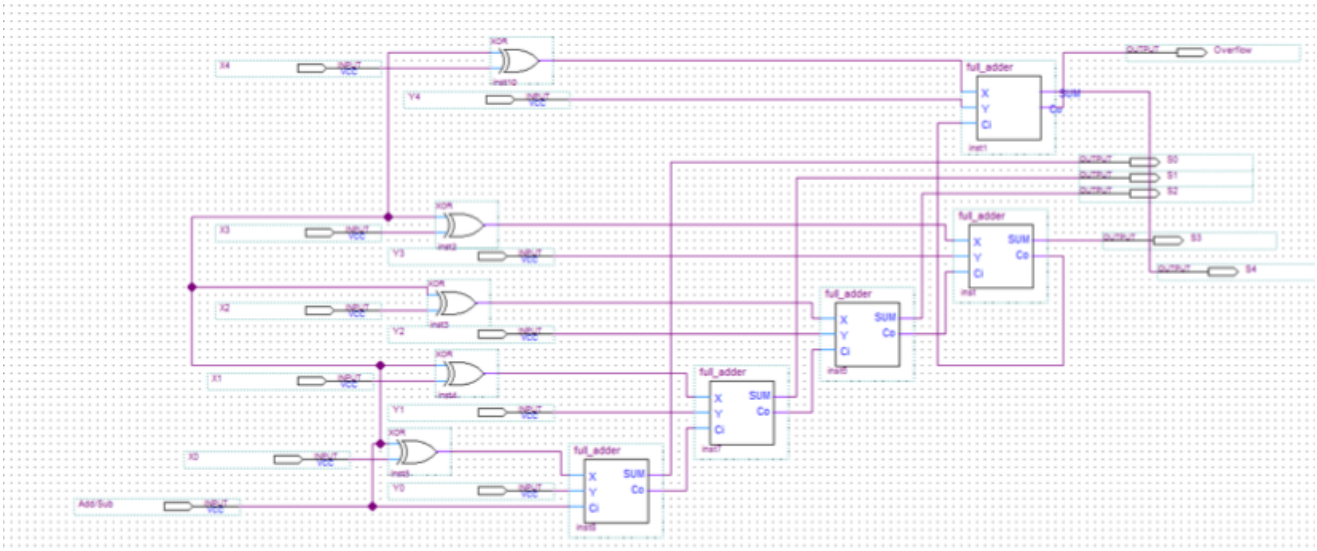
Using and-gates and full adders I made a 5-bit multiplier. If a higher order bit count is required, a 5-bit or-gate is used to detect overflow. For the multiplier the inputs are 2 registers, and the outputs are a new register value and the possible overflow.



(5-bit multiplier .bdf)

5-bit Adder/Subtractor:

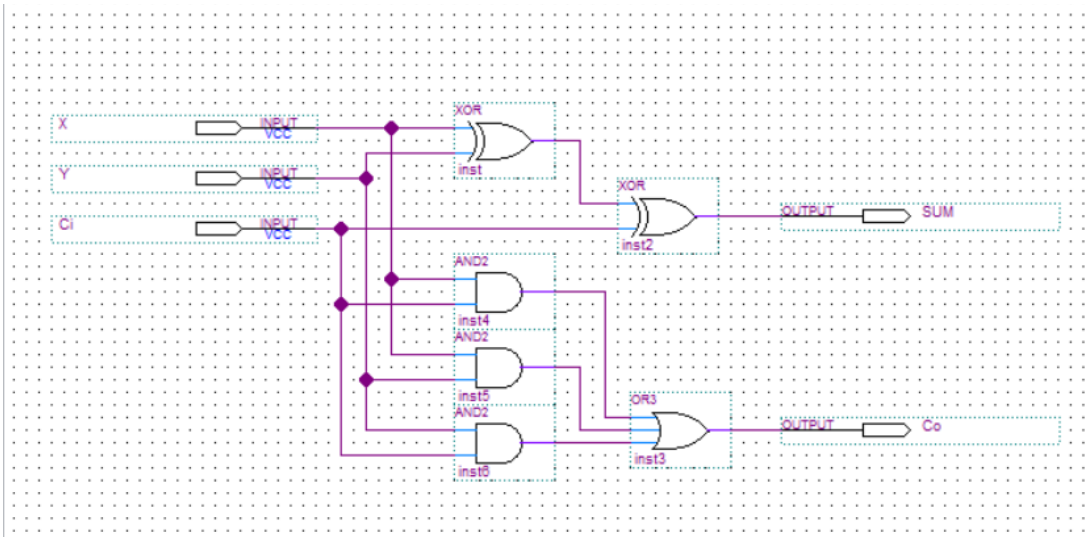
It uses a ladder of full adders to compute the addition or subtraction (XOR = 1). Overflow is checked in the last full adder. Inputs are two registers, and the outputs are a new register value and overflow.



(add/sub .bdf)

Full Adder:

A full adder uses and-gates, or-gates and xor-gates. Its inputs are two values and a carry in value, adds them together, produces an output (the sum) and the carry out.

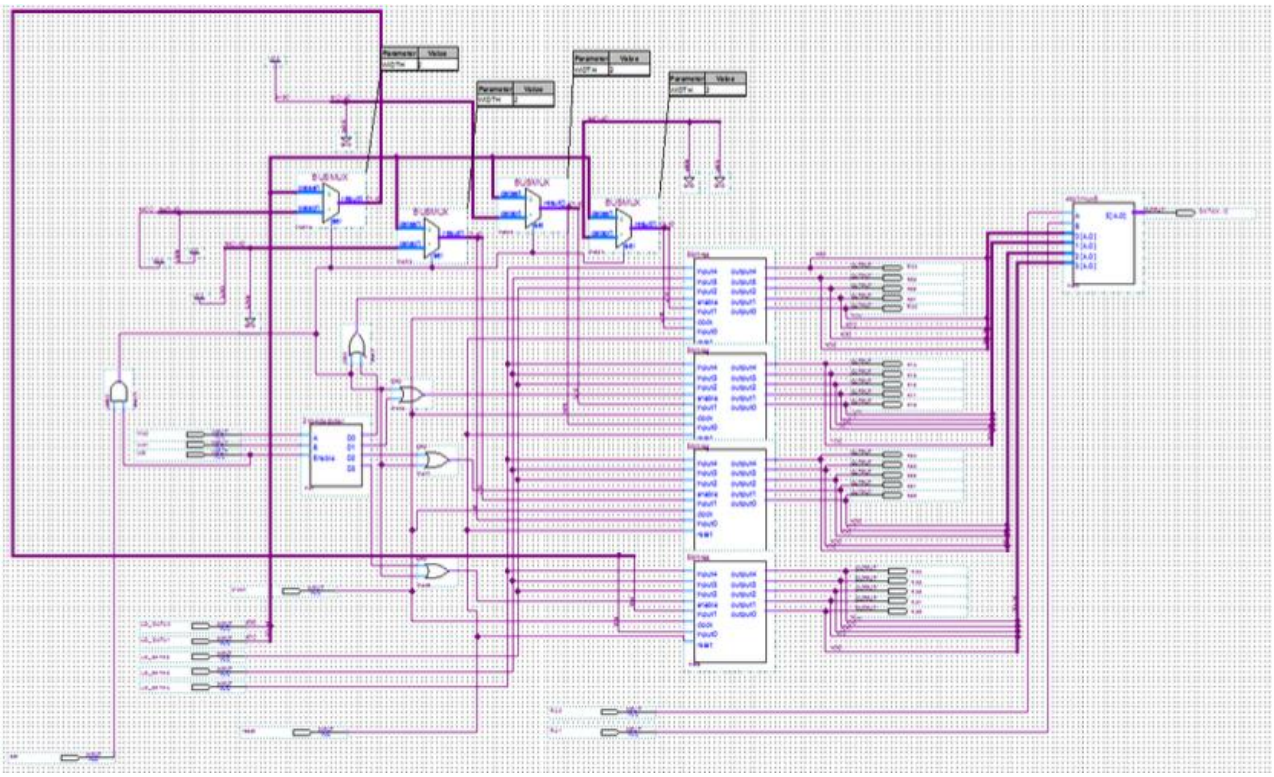


(Full adder .bdf)

5-bit Register File:

To make this register file, I used 5-bit registers, a 4 to 1 5-bit multiplexer and a 2 to 4 decoder. I also used and-gates, or-gates and some busmux for the set statement, the set statement is what handles OP000. Inputs for this register file are the register values, the write enable, the set, the reset and the write address. The outputs are the current register values and the k value.

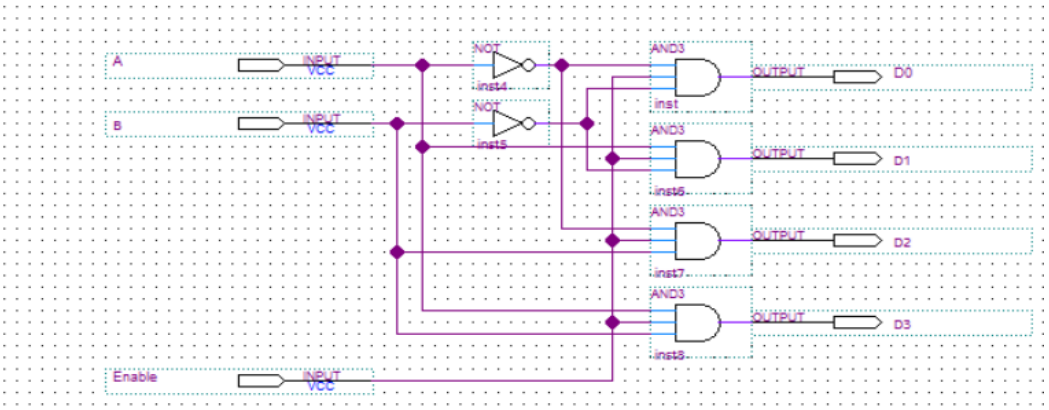
(See next page for file image)



(Register file .bdf)

2-4 Decoder:

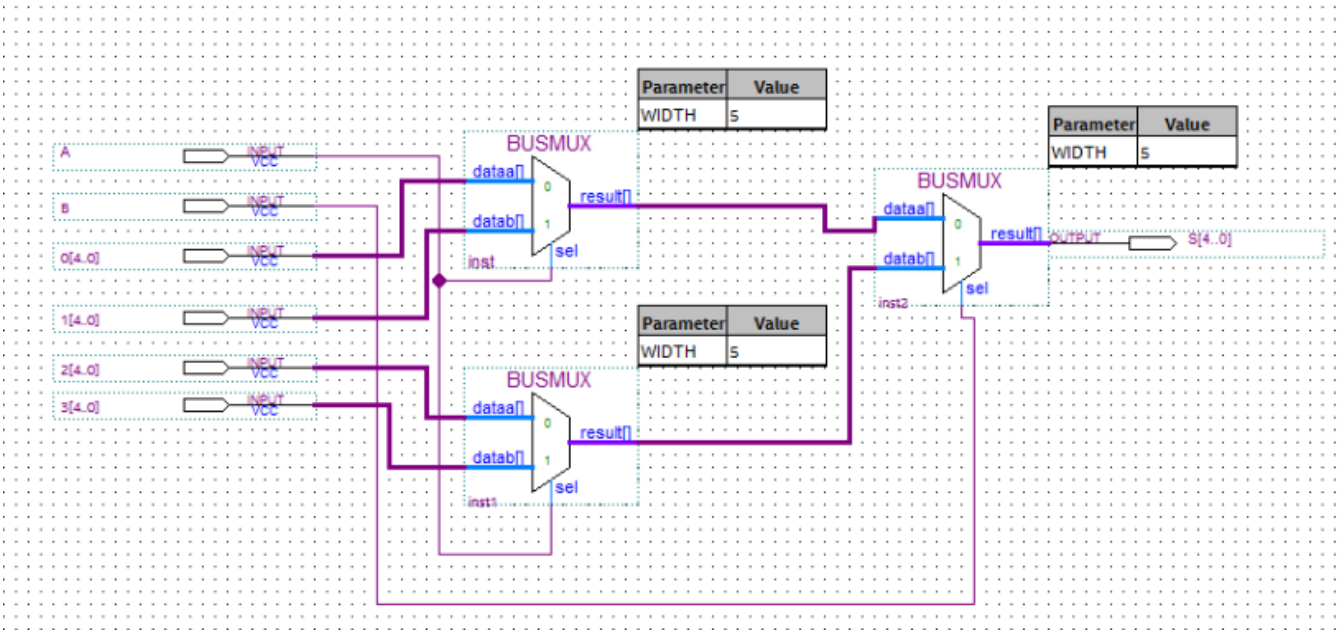
Takes in two values and an enable, outputs what register needs to be written to.



(2-4 Decoder .bdf)

5-bit 4 to 1 Multiplexer:

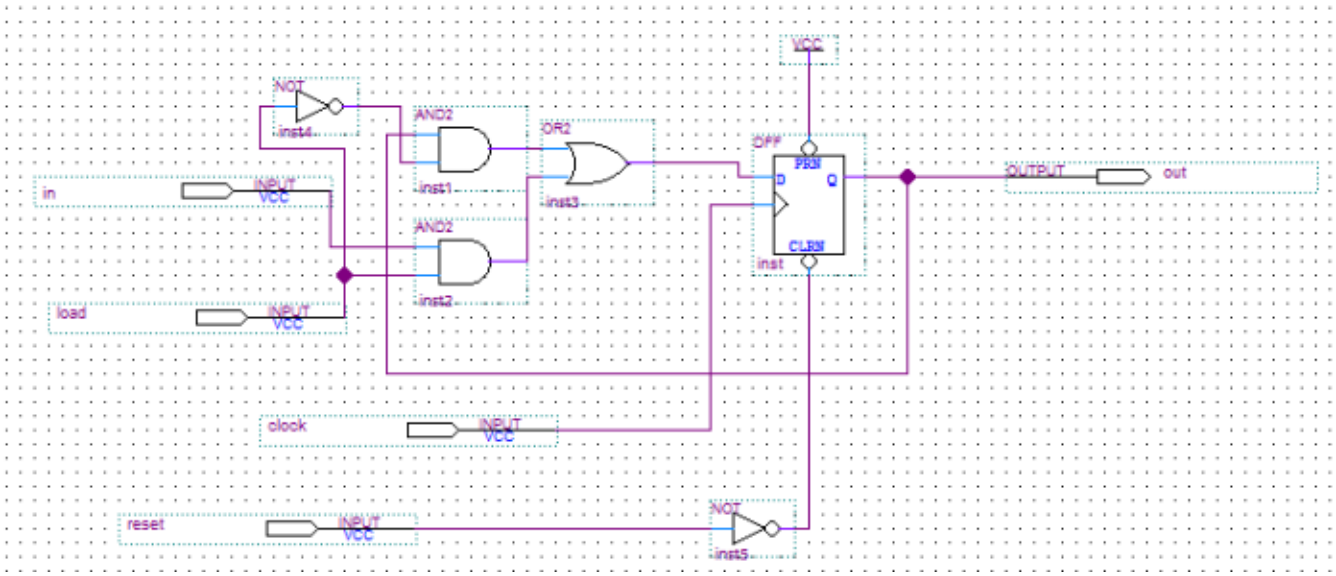
The inputs are the 4 registers and the selector, the output is the k register.



(5-bit 4 to 1 Mux .bdf)

1-bit Register:

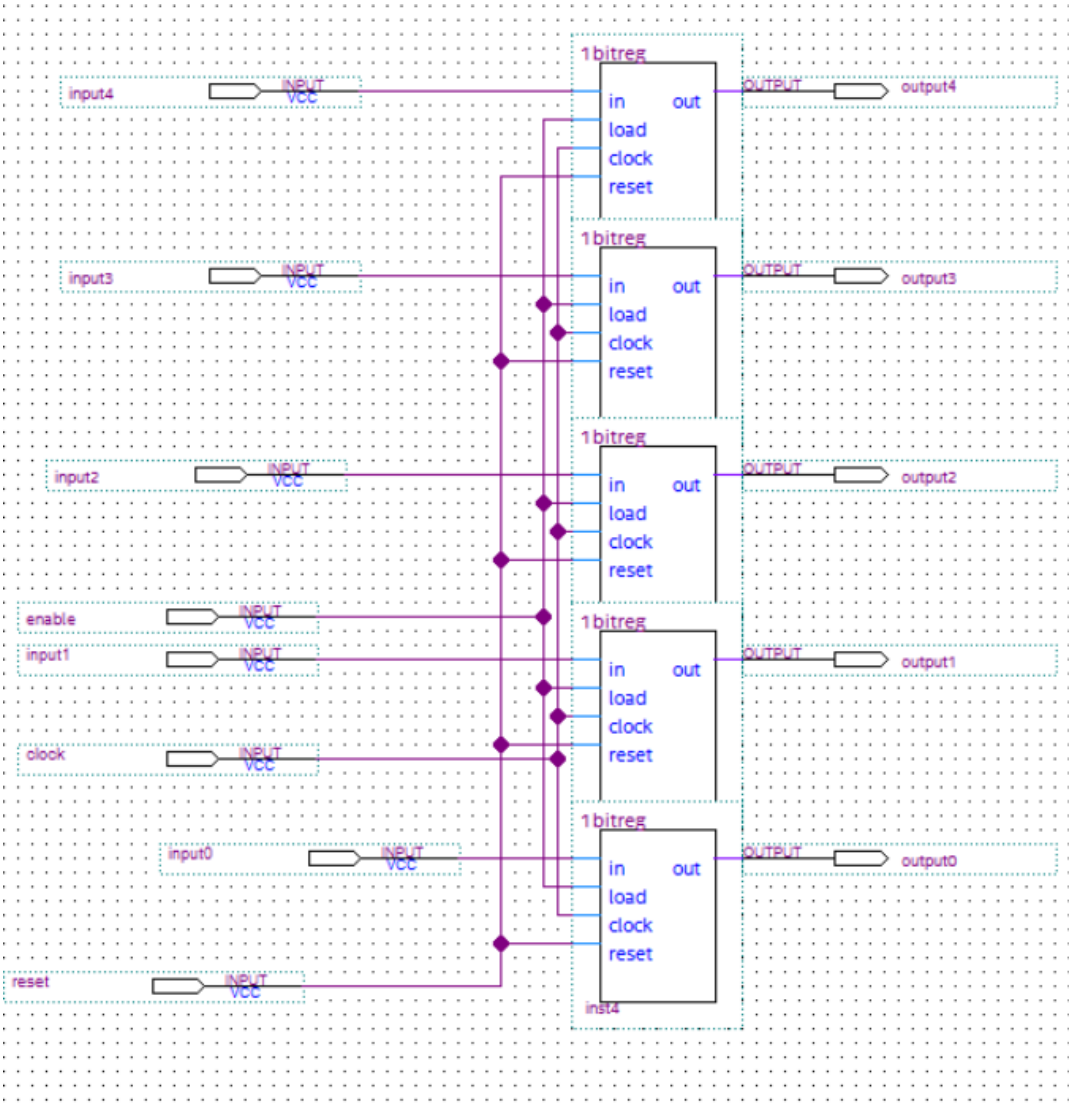
1-bit register was made using a d-flip-flop along with other gate types. It takes in an enable, a clock, a reset and a load. It outputs the single register value.



(1-bit Register .bdf)

5-bit Register:

This 5-bit register was made using 5 1-bit registers. The inputs are an enable, 5 values, a reset and a clock. The output is the current register value.



(5-bit register .bdf)

Testing:

Switch Controls: OP[2] = Switch 4

OP [1] = Switch 3

OP [0] - Switch 2

K[1] = Switch 1

K[0] = Switch 0

Perform = Switch 5

Reset = Switch 17

Example solution:

OP = 110, K = 10, Out = 3216, where R0 = 3, R1 = 2, R2 = 1, R3 = 6